Software Architecture Design and Implementation (SADI) CPT222 SP1, 2019 --- Dice Game ---

Assignment 2: Multiplayer Implementation (25 marks)

This assignment requires you to update the Dice Game from Assignment 1. Specifically, you are to implement networked multiplayer functionality using Java Sockets, Threads, and Object serialization.

1. Multiplayer Networking Functionality

You should build on top of your existing Assignment 1 functionality and should not need to change your existing GameEngine related code and other interfaces if they were fully correct in Assignment 1. A specific design structure is given below.

In order to implement the multiplayer networking functionality, you are to write the following classes:

`1) public class GameEngineClientStub implements GameEngine`

This will serve as the client (GUI) side model for your multiplayer game and since it implements the `GameEngine` interface it can be operated using your original GUI client from Assignment 1.

The purpose of this class is to perform the client-side networking functionality. This means using a `java.net.Socket` and Java serialisation (`java.io.ObjectOutputStream/ ObjectInputStream`) to send parameters to the `GameEngineServerStub` described below.
There will be one instance of this class for each client (which will run its own GUI).

`2) public class GameEngineServerStub`

This class is the server-side of the networking functionality. It should utilise a `java.net.ServerSocket` to listen for connections. Upon receiving data it should identify which method is to be called and then forward the call to your original `GameEngineImpl` from Assignment 1 (which will do the actual work). This class should also write back any data to the client via the same socket on which it received the request. **There will be one instance of this class for the whole multiplayer game** and it must be multi-threaded to be able to receive requests from multiple clients.

3) Two classes related to the `GameEngineCallback` functionality:
`ClientGameEngineCallbackServer` and `ServerStubGameEngineCallback`.

This is probably the most challenging part to implement and requires a layered architectural approach from client to server so that callbacks from the real `GameEngineImpl` (from Assignment 1) can get passed back over the network to appropriate clients.
Given the spec above you should already have a `GameEngineCallback` for your GUI from Assignment 1 that can be passed to the `GameEngineClientStub`. The `GameEngineServerStub` must also register a `GameEngineCallback` on `GameEngineImpl` (name this class `ServerStubGameEngineCallback`). To complete the whole sequence chain, you will have to write a `ClientGameEngineCallbackServer` (**one instance for each `GameEngineClientStub`**) that has a `ServerSocket` to receive callbacks across the network from the `GameEngineServerStub` and forward the results to the original `GameEngineCallback` of the client. The class `ServerStubGameEngineCallback` should serve as the client for the `ClientGameEngineCallbackServer`.

This whole structure is explained in detail in Lecture videos (especially Workshop 2) and will be discussed further during Collaborate Ultra chat sessions.

2. GUI Implementation

Each (local) player will have their own running instance of the GUI. You don't have to modify the GUI developed in Assignment 1 since most of the work for Assignment 2 is in the networking implementation. The (local) players should be able to see only their own results in the local GUI, whereas the results of all the other players should be made visible as logging info in the console (this should be shown in every client). This can be facilitated by the `GameEngineCallback` methods because they receive a *player* as a parameter.

3. Implementation Specifics

IMPORTANT: As with assignment 1, you must not change any of the original interfaces and must implement the classes detailed in section 1 of this spec document. You may implement any other additional interfaces/ classes (e.g. helper classes) as you see fit.

- For testing purposes you can run multiple copies of the GUI for multiple players on a single machine, with the server also running on the same machine. You can use the IP address 127.0.0.1/localhost for this purpose.
- For simplicity reasons, any player can "start" the game for all other connected players (i.e. "forcing" them to participate in the game). You have flexibility in how you handle this scenario, one possibility is to set the bet amount of all "forced" players to a previous or default value.
- The system must remove a Player and corresponding callback (`ServerStubGameEngineCallback`) upon player disconnection.
- You should use MVC-based implementation for your system.
- You should aim to provide high cohesion and low coupling.
- You should comment all important sections of your code (all major loops and conditional statements etc.).
- You should take care with thread synchronization and ensure there are no deadlocks or race conditions.
- You should handle server-side exceptions with a console error messages, but do not need to handle them in the GUI i.e. you can assume a correct running server when using the multiplayer engine. *You should also provide basic server-side console logging to demonstrate players' connection/disconnection status.*

4. Submission Instructions

- You are free to refer to textbooks and notes, and discuss the design issues (and associated general solutions) with your fellow students on Blackboard; however, the assignment must be your own individual work.
- Where you do make use of other references, please cite them in your work. Note that you will only be assessed on your own work so the use of third party code or packages is not advised.
- The source code for this assignment should be submitted as a .zip file through the Assignments section of Canvas. You need to submit a complete compiled **Eclipse project that includes source-code for both Assignment 1 and Assignment 2 i.e. the submission must be self-inclusive so that it can be executed directly by the marker**.

Additional requirements:

- Please name your client-side driver class A*ss2Driver.java*, and place it into *package au.edu.rmit.cpt222.driver;*
- Please name your server-side driver class *Server.java* and place it into *package au.edu.rmit.cpt222.test.ass2;*
- Please provide a brief *readme.txt* file explaining important network-related considerations (e.g. your networking setup in terms of ports used), and other relevant details, if any.

Due Date: see Syllabus (in Canvas). Late submissions are handled as per usual RMIT regulations - 10% deduction (2.5 marks) per day. You are only allowed to have 5 late days maximum.